



Implementando o frontend CRUD do projeto integrado

Chegamos ao último módulo da disciplina Prática Integradora. Agora, implementaremos o frontend de nossa aplicação CRUD, unindo as telas ao backend construído nos módulos anteriores. Neste módulo aproveitaremos tanto os conceitos já abordados sobre a biblioteca React Js, além de outras libs, como Axios e Material UI, por exemplo, como telas já codificadas. Para começar, vamos criar um novo projeto React, assim como a estrutura de pastas com a qual já estamos familiarizados (montamos essa estrutura no módulo 13). Sobre a aplicação em si, e considerando a quantidade de telas e códigos envolvidos, não veremos, aqui, todos os códigos, mas passaremos pelos principais fluxos e lógicas, montando a estrutura para que você consiga, unindo todos esses materiais, concluir a codificação de todo o frontend. Além disso, disponibilizarei alguns códigos ao final, contendo, em detalhes, o que for apresentado ao longo desse módulo.

De início, e apenas para relembrar, teremos a seguinte a estrutura de pastas:

- /src

  /api (instância Axios para conexão com o backend)

  /components (componentes reaproveitáveis, como a appbar)



/pages (páginas/telas da aplicação)



/routes (rotas da aplicação)

/services (serviços da aplicação)

Em relação às bibliotecas, utilizaremos as seguintes:

- @emotion/react
- @emotion/styled
- @mui/icons-material
- @mui/material
- @mui/x-data-grid
- axios
- react-router-dom

Dentro da estrutura de diretórios dos códigos de frontend, só não falamos ainda sobre a pasta "services". Nela guardaremos os serviços de nossa aplicação, ou seja, os componentes funcionais responsáveis por prover serviços como o login, por exemplo. Separar esses códigos melhora o desacoplamento, uma vez que manteremos em lugares distintos os componentes com função visual dos componentes com função de serviços e lógica, entre outros. Em nossa aplicação, criaremos um script dentro dessa pasta chamado LoginService.js . Ele será responsável por receber, da tela de login, o usuário e senha e acessar, via Axios, o backend a fim de validar as credenciais. Com as mesmas validades, ele armazenará no

LocalStorage o jwt recebido, que precisará ser recuperado e enviado a cada nova solicitação ao backend (tendo em vista a a configuração de segurança que implementamos em nossa API).

Aproveitando que falamos dele, vamos entender a utilidade do LocalStorage. Tal recurso permite o armazenamento em cache, no navegador, de dados e deve ser usado em ocasiões onde precisamos guardar pequenas quantidades de informação, das quais precisaremos fazer uso na aplicação, sem a necessidade de consultarmos o banco de dados, por exemplo, para obtê-las. É muito utilizado para o mesmo propósito que o nosso: guardar dados do usuário logado, em nosso caso representado pelo JWT, chamados também de dados de sessão. Sobre esse último termo, é importante ter em mente que esses dados armazenados no LocalStorage são válidos/existem apenas enquanto o usuário estiver com a janela do navegador aberta. A partir do momento em que ele encerra o navegador, os dados armazenados são perdidos. Também é importante ter em mente que esses dados armazenados precisam ter um período de validade/expiração, para evitar que outros usuários, utilizando um mesmo computador, aproveitem uma sessão anterior para obter acesso indevido à aplicação.

Daqui em diante veremos os fluxos de nossa aplicação, assim como as funcionalidades que precisam ser implementadas. Começaremos pelo Login:

A tela de login deverá ser composta por um formulário com dois campos: e-mail e senha. Essa tela ficará armazenada na pasta Login, dentro da pasta pages. O formulário, ao ser submetido, usará o serviço LoginService, que retornará o JWT, caso as credenciais sejam válidas. A seguir, deverá ser armazenado em LocalStorage o JWT e a data/hora de início da da sessão. Aqui podemos usar uma

string JSON, com duas chaves: a primeira contendo o valor do JWT e a segunda a data/hora. A seguir, o usuário será levado para a página Home. Tal página poderá conter a listagem de tarefas ou a listagem de projetos, por exemplo.

Em termos de telas e componentes, nossa aplicação usará a AppBar vista em módulos anteriores, acrescida de um menu do tipo Drawer, que permitirá ao usuário navegar entre as telas disponíveis. Considerando que esse componente de AppBar e Menu será carregado/renderizado em todas as telas, incluiremos nele um mecanismo de verificação da validade da sessão, recuperando os dados armazenados no LocalStorage. Caso a data/hora de início da sessão mais um quantidade X (60, por ex) de segundos tiver sido alcançada, os dados da sessão serão deletados do storage e o usuário não conseguirá mais navegar, sendo necessário a realização de um novo login para isso. Veja abaixo um esboço da função de submissão do form de login com a lógica de armazenamento de dados mencionada acima:



```
const handleSubmit = async () => {

  //Modifica o state para nao exibir a msg de erro
    //(limpa a msg qdo tenha sido exibida anteriormente)
  setShowRenderErrorMessage(false);

  //Modifica os states para exibir o botao de "carregando"
  if (!loading) {
    setSuccess(false);
    setLoading(true);
  }

  //Faz a chamada do servico de Login passando usuario e senha
  var userData = await LoginService(userName, password);

  //Exibe no console os dados do usuario retornado no servico
    //acima ou undefined caso o mesmo nao exista
  console.log('userData: ' + JSON.stringify(userData));

  // Se nao encontrou o usuario ou se os dados estao incorretos
  if (!userData) {

    //Modifica o state para exibir a msg de erro de credenciais invalidas
    setShowRenderErrorMessage(true);
    setErrorMessages({ name: "credenciais", message: errors.credenciais });

    //Modifica os states para inibir o botao de "carregando"
    setSuccess(false);
    setLoading(false);

  } else {

    //Modifica os states para inibir o botao de "carregando"
    setSuccess(true);
    setLoading(false);

    //armazena os dados do usuario (nesse caso, o username)
     //e a data atual para controle da expiracao da sessao no
     //local storage
    //  CUIDADO: a senha nao deve ser armazenada
    localStorage.setItem('authUser', JSON.stringify(
      {jwt: userData["jwt-token"], exp:Date.now()}
    ));

    //navega para a Home
    navigate("/")
  }
};
```

O código está comentado, explicando cada um dos passos realizados. Como dito, trata-se de um esboço, que precisará de pequenas mudanças de sua parte, para se integrar totalmente ao backend – leia-se: o retorno do LoginService, via objeto userData, será composto pelo JWT. Logo, você precisará alterar o objeto userData.username para algo como userDate.jwt . Além disso,

precisará armazenar o conteúdo do JWT nesse objeto no LoginService, e enviá-lo de volta para o método de login. Repare no código que há dois states, chamados success e loading. Esses states são usados para exibir ao usuário uma informação visual que o permita entender que sua ação, nesse caso a submissão dos dados de login, está sendo processada. Devemos, inclusive, fazer esse tipo de tratamento em toda a aplicação, em todos os lugares onde uma ação do usuário faz com que um processamento seja iniciado. É importante para ele ter conhecimento disso, para uma melhor usabilidade, e para que não ache que nada aconteceu ao clicar em determinado botão ou entrar numa tela em branco, sem que saiba que algo "por trás" está acontecendo. Logo, utilize esse recurso em toda a aplicação, seguindo a mesma lógica aplicada aqui. Ainda sobre o JWT armazenado na sessão, você precisará dele a cada requisição que realizar ao backend, tendo em vista que implementamos o controle de segurança na API. Tal dado deverá ser enviado via Axios, no header da requisição, como bearer token.

Sobre o controle do tempo de sessão, que, conforme dito, será feito no componente Header, uma vez que o mesmo é renderizado em todas as telas da aplicação, uma ideia de implementação é utilizar o Hook useEffect, ouvindo/monitorando uma variável que informe que o usuário mudou de tela. Para isso, para verificar se o usuário mudou de tela, utilizaremos um recurso da biblioteca react-router-dom (que também utilizaremos para montar as rotas de nossa aplicação), chamado useLocation. Logo, a cada alteração obtida via useLocation, podemos chamar uma função que verifique a data/hora de início da sessão e identifique se passou um tempo X desde o seu início. Esse tempo também pode ser configurado no próprio método de checagem, ou armazenado em variável de ambiente ou até mesmo no localstorage. Veja abaixo uma ideia de como ficaria o

componente Header com o Hook useEffect implementando a lógica acima:



```
//Outros imports …

import { useNavigate, useLocation } from 'react-router-dom';
import CheckAuth from './CheckAuth';

import { ListaMenu } from '../routes/RoutesNavegacao';

{/*Componente para exibicao da Barra no topo da aplicacao (appbar) */}
const Header = () => {
  const location = useLocation();

  //O hook abaixo sera executado sempre que o usuario mudar de pagina.
  //  A cada mudanca, sera verificado, no localstorage,
      //se o token/autenticacao expirou.
  //  Em caso positivo, o usuario sera deslogado do sistema
  useEffect(() => {

    //Chama o Componente Funcional para
    //  verificar se a autenticacao expirou.
    //  Em caso positivo, leva o usuario para
    //  a tela de login
    if(!CheckAuth(location))
      navigate("/login")

  }, [location]);

//restante do código do componente Header ...
```

Dentro do useEffect é chamado o componente CheckAuth. Sua função é recuperar os dados da sessão, verificar se o seu tempo de duração excedeu e, em caso positivo, “deslogar” o usuário, limpando o localstorage. Veja abaixo sua implementação:

```
const CheckAuth = (location) => {
  if (localStorage.getItem("authUser")) {
    const jwt_Token_decoded = JSON.parse(localStorage.getItem("authUse

    //Armazena o tempo da sessao em milisegundos (60000 ml = 1 min)
    const tempoDuracaoSessao = 60000;

    //Se a data/hora atuais (em milisegundos)
      //for maior que a data/hora setada para a sessao,
    //  limpa o storage e desloga o usuario.
    //  Com o localStorage limpo, o usuario nao
      //conseguira mais navegar pra outras paginas, sem antes
    //  se logar novamente.
    if (Date.now() >
      parseInt(jwt_Token_decoded.exp + tempoDuracaoSessao)  ) {
      localStorage.clear();
      return false;
    }else{
      return true;
    }
  }
}

export default CheckAuth;
```

Sobre o tempo de duração da sessão, perceba que há uma variável constante onde foi definido o tempo de 1 minuto. Essa parte do código poderia ser melhorada, retirando desse método tal informação, por exemplo.

Na sequência de uso da aplicação, após o login o usuário é levado para uma página inicial. Tal página pode ter diferentes conteúdos: uma página em branco, sem informações; um dashboard com dados resumidos sobre projetos e tarefas; a página/inicial de tarefas ou projetos, com sua listagem; etc. Em relação à navegação, a mesma se dará através de um menu do tipo drawer, acessível a partir do menu do tipo "hamburguer". As opções do menu e também o controle de navegação são providos pelo componente react-router-dom, onde há ainda um controle de rotas privadas e públicas: as primeiras são as telas que só podem ser acessadas após o usuário realizar o login; as demais, as que não dependem de login – que,

inicialmente, em nosso caso, se restringe à própria página de login. Veja a seguir um exemplo de componente contendo rotas privadas e públicas com a biblioteca supra mencionada:

```
{/*Componente react-router-dom contendo as rotas da aplicacao -
privadas/publicas */}
const RoutesNavegacao = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route
          path="/"
          element={
              <PrivateRoute>
                  <Header />
                  <Home />
              </PrivateRoute>
          }
        />
        <Route
          path="/tarefa"
          element={
              <PrivateRoute>
                  <Header />
                  <ListarTarefa />
              </PrivateRoute>
          }
        />
        <Route path="/login" element={<Login />} />
        <Route path="*" element={<Navigate to="/" />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Nesse cenário, o componente PrivateRoute verifica se o usuário está logado, coletando tal informação a partir do local storage, e permite ou não o acesso às rotas “filho” (os componentes de tela por ele englobados no código acima). Note que uma das rotas privadas é a nossa tela CRUD ListarTarefa. Abaixo você poderá ver um esboço desse componente, que, embora funcional, também pode ser melhorado por você no futuro:

```
import { Navigate } from 'react-router-dom';

const PrivateRoute = ({ children }) => {
  const authUser = JSON.parse(localStorage.getItem('authUser')) || false;
  //console.log('authUser: ' + JSON.stringify(authUser));

  if (!authUser) {
      // Se nao encontrar o usuario no AsyncStorare
       //(usuario nao logado) redireciona pra pag. de Login
      return <Navigate to="/login" />
  }

  // Encontrando o usuario, permite a navegacao pros demais componentes
  return children;
}

export { PrivateRoute };
```

Por fim, além das telas e funções discutidas até aqui, relacionadas ao login, controle de sessão e integração com o usuário/usabilidade, temos as telas CRUD em si, cujas principais páginas são as de projeto e tarefa. Sobre essa última, já a implementamos no módulo 10, cujo repositório pode ser acessado pelo link:

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo10/template-materialui/src/pages/tarefa **(Acesso em 20/10/2022)**

Na ocasião dessa implementação, não ligamos o frontend ao backend, deixando todos os dados "mockados" diretamente na tela. Em nossa implementação final, utilizaremos a API recém desenvolvida, utilizando no front o Axios para fazer a integração entre as partes. Nesse sentido, toda a lógica já aplicada pode ser mantida, mudando apenas a fonte dos dados da tela. Seguindo esse raciocínio, o mesmo deverá ser aplicado ao CRUD de projetos. Veja abaixo a listagem de tarefas refatorada para utilizar a API:

```
// …
const [tarefas, setTarefas] = useState();

useEffect(() => {
  getTarefas();
}, []);

const getTarefas = async () => {
  InstanciaAxios.get(
        `/tarefa`,
        {headers: { Authorization: `Bearer ${token.jwt}` }}
      )
      .then(result => {
        console.log('result data: ' + JSON.stringify(result.data));
        setTarefas(result.data);
      });
}
// ...
```



Na implementação mockada, definimos um array local que alimentava a tabela contendo a listagem de tarefas. Nessa nova implementação, recuperamos as tarefas existentes no banco, acessando a API com o Axios e armazenando os dados no state Tarefas. Como esse state também é um array, só precisaremos modificar o laço *for* que exibe as linhas da tabela para alterar os nomes das chaves do array. Além da listagem, também precisaremos alterar a criação, edição e deleção de tarefas. Veja como fica a criação de tarefas. A partir dessa implementação, você poderá implementar a edição, seguindo a mesma lógica vista aqui.

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo16/frontend **(Acesso em 20/10/2022)**

Sobre o código e seu fluxo, cabe fazer alguns comentários:

- Através do Hook useEffect, na renderização inicial do componente, são chamados alguns métodos que conectam ao backend e recuperam os dados para população dos componentes select contidos na aplicação;

- O método handleSalvar envia os dados da tarefa, contidos nos states, para o backend. Ao final, em caso de sucesso no envio, ele chama o método responsável por atualizar a listagem de tarefas (que ele recebeu do componente ListarTarefas via props);

- Foi necessário um tratamento específico pras datas, para facilitar sua compatibilidade com o backend. Nesse sentido, foi utilizado o formato de data em milissegundos.

Como dito, a partir dessa implementação, você poderá desenvolver – conforme pedido na Atividade Prática – a edição de tarefas. O mesmo se aplica às demais telas da aplicação, cuja lógica, por se tratarem de telas CRUD, poderá seguir a mesma aplicada às tarefas.

Após vermos os detalhes e fluxos para implementação do frontend de nosso projeto aplicado, chegamos ao final do nosso último módulo, cujo conteúdo complementa os demais, vistos ao longo dessa disciplina, e que nos permitiram obter a base de conhecimentos teóricos e também práticos, para a construção de uma aplicação fullstack completa, utilizando uma API Rest, implementada com o framework Spring Boot, cujo front foi construído com a lib React Js, fazendo parte ainda dessa arquitetura o banco de dados Postgres.



Atividade Extra

A biblioteca Material UI disponibiliza vários componentes de interface, entre eles um Data Grid que permite a exibição de dados tabulares, sendo um componente tanto visual quanto funcionalmente superior às tabelas simples. Sugiro que veja mais sobre tal componente acessando o link abaixo:

https://mui.com/pt/x/react-data-grid/ **(Acesso em 20/10/2022)**

Referência Bibliográfica

MORGAN, Joe. **How to code in React.js**. New York City, NY: Digital Ocean, 2021.

Atividade Prática Módulo 16

Título da Prática: Refatorando a Edição de Tarefa.

Objetivos: Refatorar a tela de Edição de Tarefa, implementando a integração com o backend.

Materiais, Métodos e Ferramentas: VS Code, Docker.

Atividade Prática

Durante o módulo 16, tratamos da codificação do frontend de nossa aplicação integrada. Nesse sentido, refatoramos o CRUD de tarefas, trocando o consumo de dados mockados pelos dados existentes no banco, através da integração com o backend. Nessa atividade prática você deverá implementar a tela de edição de tarefas, refatorando-a, a exemplo do que foi feito ao longo do módulo com a tela de criação de tarefa – que você deverá utilizar aqui como modelo.

Gabarito Atividade Prática

Ao final dessa atividade, o aluno deverá ter codificado a tela de Edição de Tarefa, totalmente funcional – ou seja, permitindo a edição dos dados de uma tarefa, de forma integrada com o backend da aplicação. Como gabarito, podendo ser utilizado para fins de comparação/validação pelo aluno, um exemplo de código contendo a resolução do exercício está disponível no link abaixo.

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo16/atividade-pratica-gabarito **(Acesso em 10/10/2022)**

Ir para exercício